合肥工業大學

系统软件综合设计报告
操作系统分册

设计题目	分段存储管理系统
学生姓名	袁焕发
学　　号	2019217769
专业班级	物联网工程 19-2 班
指导教师	田卫东
完成日期	2021.01.06

1. 课程设计任务、要求、目的

1.1. 课程设计任务

建立一个基本分段存储管理系统模型，演示分段存储系统工作流程。

1.2. 课程设计目的和要求

首先分配一片较大的内存空间，作为程序运行的可用存储空间；建立应用程序的模型，应该包括相应的分段描述与存储结构；建立进程的基本数据结构及相应算法；建立管理存储空间的基本存储结构。建立管理分段的基本数据结构与算法；设计存储空间的分配与回收算法；提供信息转储功能，可将存储信息存入磁盘，也可从磁盘读入。

2. 开发环境

Windows 11、DEVC++ 5.12

3. 相关原理及算法

3.1. 分段地址结构

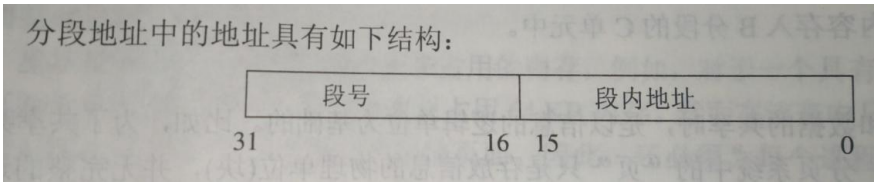


图 1 分段地址结构

将用户作业的逻辑地址空间划分成若干个大小不等的段（由用户根据逻辑信息的相对完整来划分）。各段有段名（常用段号代替），首地址为 0。

3.2. 段表地址映射

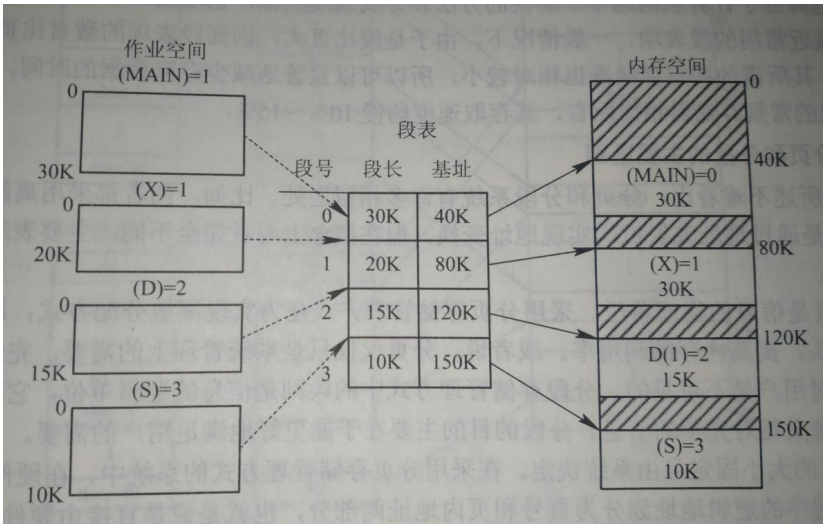


图 2 段表地址映射

段表记录了段与内存位置的对应关系，段表保存在内存中，段表的基址及长度由段表寄存器给出。

段表始址	段表长度

访问一个字节的数据/指令需访问内存两次（段表一次，内存一次），逻辑地

址由段和段内地址组成。

3.3. 地址变换

段表(起)始地址+段号*段表项长度,得到该表项在段表中的位置,访问它即可得到段长和段始址。检查段内地址是否超过段长,等于或大于产生中断,否则进入下一步。该段始址+段内位移,得到欲访问数据的最终地址,访问它即可得到数据。

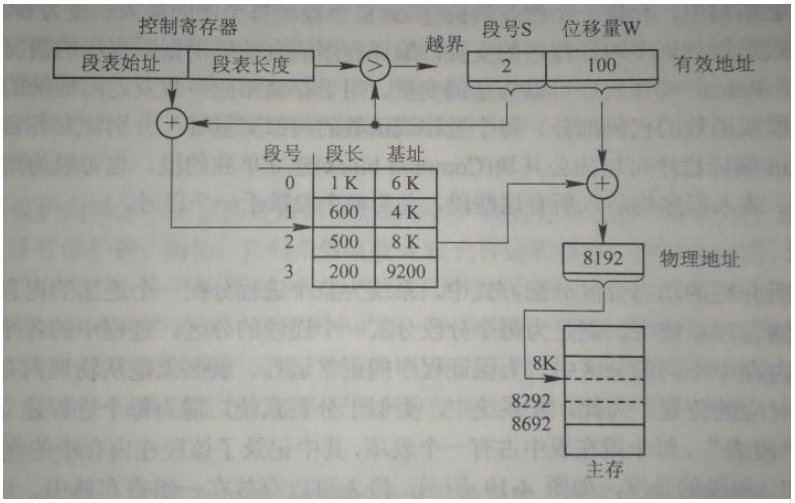
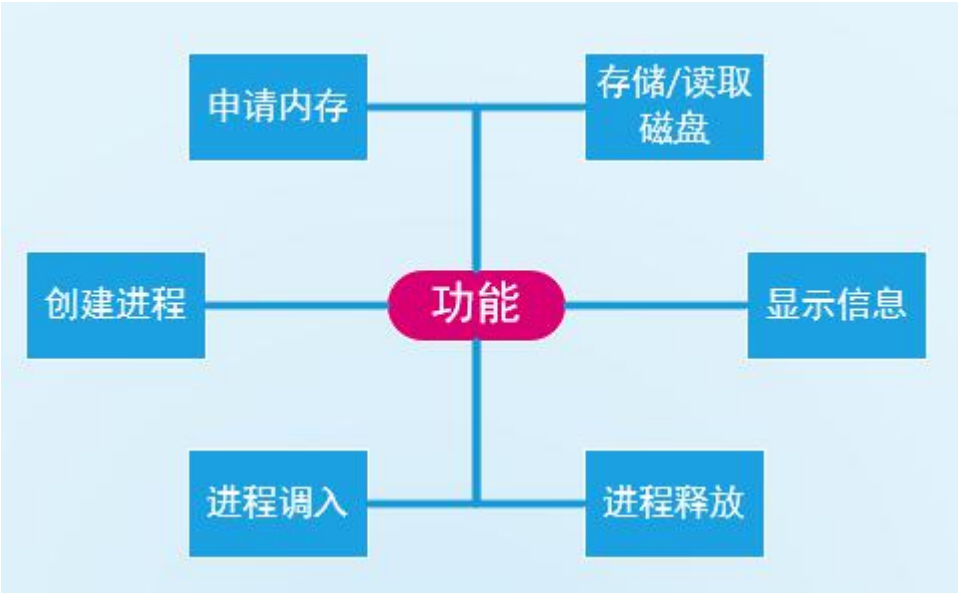


图 3 段表地址变换

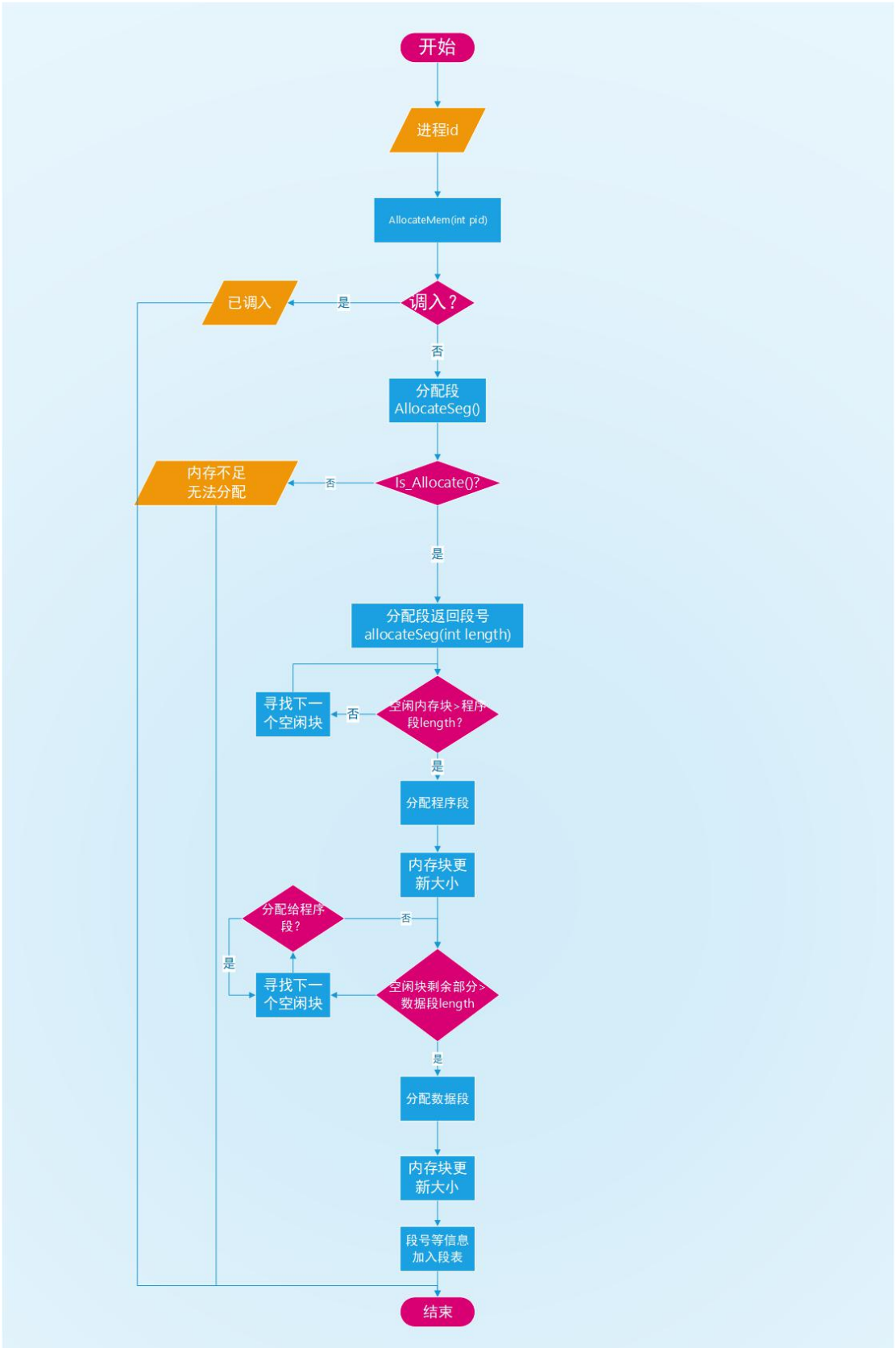
4. 系统结构和主要的算法设计思路

4.1. 算法思想

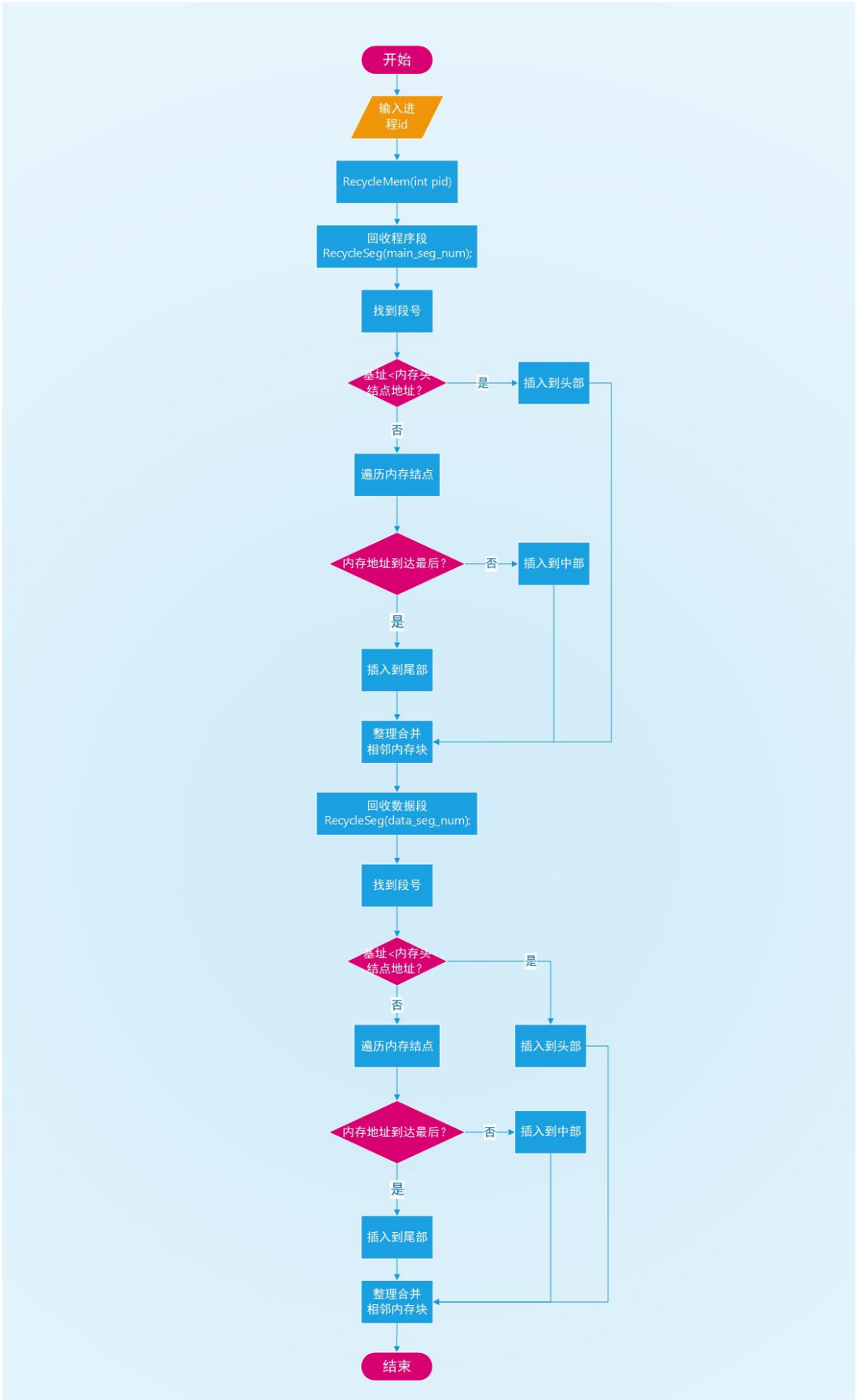
首先利用申请动态内存来作为分段存储的空间，然后模拟创建进程，用于作为分段存储的对象，然后编写段的数据结构，以及段表的数据结构，来实现最关键的分段存储功能，在此基础上要对已经分配好的空间中的信息进行访问，删除和回收空间，最终以链表作为存储结构，以结构体作为段和段表的结构。



分配流程


开始
进程id
AllocateMem(int pid)
调入？
是
已调入
否
分配段
AllocateSeg()
Is_Allocate()?
否
内存不足
无法分配
是
分配段返回段号
allocateSeg(int length)
空闲内存块>程序段length?
否
寻找下一个空闲块
是
分配程序段
内存块更新大小
分配给程序段?
否
是
寻找下一个空闲块
空闲块剩余部分>数据段length
是
分配数据段
内存块更新大小
段号等信息加入段表
结束

内存回收流程


开始
输入进程id
RecycleMem(int pid)
回收程序段
RecycleSeg(main_seg_num);
找到段号
基址<内存头结点地址？
是
插入到头部
否
遍历内存结点
内存地址到达最后？
否
插入到中部
是
插入到尾部
整理合并相邻内存块
回收数据段
RecycleSeg(data_seg_num);
找到段号
基址<内存头结点地址？
是
否
遍历内存结点
插入到头部
内存地址到达最后？
否
插入到中部
是
插入到尾部
整理合并相邻内存块
结束

5. 程序实现---主要数据结构

```
struct Program //进程
{
    int pid; //  进程 id
    struct Program *next; //  下一个进程的 id
    int main_seg_num;   //程序段号
    int main_length; //  程序段长
    int data_seg_num;//  数据段号
    int data_length;//数据段长

};

struct Segtable//段表
{
    int seg_num; //  段号
    int seg_length; //  段长
    int base_addr; //  基址
    struct Segtable *next; //  下一个段指针
};


struct Memory //内存
{
    int addr;     //  始址
    int length; //  长度
    struct Memory *next; //  下一个
};
```

段、段表、内存空间均为结构体类型，每个单元之间采用链式连接。

6. 程序实现---主要程序清单

（本人部分）

```
//创建进程
struct Program *CreateProgram(int pid, int main_length, int data_length) {
    struct Program *newPro;
    newPro = (struct Program*)malloc (sizeof(struct Program));
    //newPro = new program;
    newPro->pid = pid;
    newPro->main_length = main_length;
    newPro->main_seg_num = -1; //  初始化
    newPro->data_length = data_length;
    newPro->data_seg_num = -1; //  初始化
    newPro->next = NULL;
    return newPro;
```

```
}

// 分配内存
void AllocateMem(int pid) {
	struct Program *index;
	index = pro_head;
	if (index != NULL) {
		for (; index != NULL; index = index->next) {
			if (index->pid == pid) {
				if (index->main_seg_num == -1) {
					// 分配段
					AllocateSeg(index);
				}
				else {
					printf("已被调入\n");
				}
			}
			//else{
			//printf("无此进程\n");
			//break;
			//}
		}
	}
}

// 进程分配段
void AllocateSeg(struct Program *pro) {
	struct Program *tmp;
	tmp = pro;
	int main_length = tmp->main_length;
	int data_length = tmp->data_length;
	// 判断可以分配
	if (Is_Allocate(main_length, data_length)) {
		if (main_length > 0) {
			int main_num = allocateSeg(main_length);
			tmp->main_seg_num = main_num;
		}

		if (data_length > 0) {
			int data_num = allocateSeg(data_length);
			tmp->data_seg_num = data_num;
		}

	}
```

```
    else {
        printf("内存不足无法分配\n");
    }
}
// 可分配
bool Is_Allocate(int length1, int length2) {
    bool flag = false;
    bool alloc1 = false; // 分配程序段，初始化标志 false
    bool alloc2 = false; // 分配数据段，初始化标志 false
    struct Memory *index;
    index = mem_head;
    int selected_addr = -1; // 已经选择了的节点
    for (; index != NULL; index = index->next) {
        // 可用内存块长度大于程序段长度
        if (index->length >= length1) {
            alloc1 = true;//分配
            int sub = 0;
            sub = index->length - length1;
            //分配完程序段，可用内存块剩余长度还大于数据段
            if (sub >= length2 && !alloc2) {
                alloc2 = true;// 分配数据段
                break; // 退出
            }
            else {
                selected_addr = index->addr; // 该地址已经分配给程序段了
                break; // 退出
            }
        }
    }
    //另找内存块分配数据段
    if (!alloc2) {
        index = mem_head;
        for (; index != NULL; index = index->next) {
            if (index->length >= length2 && index->addr != selected_addr) {
                // 空闲内存块大于数据段且没有被程序段占用
                alloc2 = true;
                break;
            }
        }
    }

    //分配测试成功
    if (alloc1 && alloc2) {
        flag = true;
```

```
    }
    return flag;
}

// 分配段表
int allocateSeg(int length) {
    struct Segtable*item;
    item = (struct Segtable*)malloc(sizeof(struct Segtable));
    item->seg_length = length;//段长
    //分配内存
    item->base_addr = allocateMem(length);
    item->seg_num = current_seg_num;
    item->next = NULL;
    int renum = -1; // 返回值
    current_seg_num += 1; // 段号加一

    struct Segtable*p;// 加入段表
    bool flag = false; // 是否分配标记
    p = seg_table_head;
    if (p == NULL) {   // 未分配
        seg_table_head = item;//分配
        renum = item->seg_num;
    }
    else { // 插入尾部
            // 寻找值为-1 的进行分配
        for (; p != NULL; p = p->next) {
            if (p->seg_length == -1) {
                // 该条没有使用
                renum = p->seg_num;
                p->base_addr = item->base_addr;
                p->seg_length = item->seg_length;
                current_seg_num -= 1;
                free(item); // 删除该节点
                flag = true; // 已经分配好段表
                break;
            }
        }
        // 如果上述都未分配,则在尾部添加一个节点
        p = seg_table_head;
        if (!flag) {
            // 找到尾部
            if (p->next == NULL) {
                p->next = item; // 插入尾部
                renum = item->seg_num;
```

```
                }
                else {
                    // 定位到尾部
                    for (; p->next != NULL; p = p->next) {
                    continue;
                    }
                    p->next = item;
                    renum = item->seg_num;
                }
            }
        }
        return renum; // 返回段表段号
}

//分配内存
int allocateMem(int length) {
        // 遍历空闲内存块，找到第一个>=待分配长度的
        struct Memory*idle, *pre;
        idle = mem_head;
        pre = mem_head; // 前驱指针
        int base_addr; // 起始地址
                        // 遍历开始
        for (; (idle->length < length) && idle != NULL; idle = idle->next) {
            pre = idle; // 保留前驱指针
        }
        if (idle == NULL) {
            //到内存最后了
            printf( "无法分配\n");
        }
        else {
            //可分配情况
            if (idle->length == length) {
                //恰好放入，删除该空闲块，合并已用内存
                pre->next = idle->next;
                base_addr = idle->length;
                free(idle); // 删除节点
            }
            else {
                // 无法填满该内存块
                base_addr = idle->addr;
                idle->addr = base_addr + length;
                idle->length = idle->length - length; // 更新空闲块的大小
            }
        }
```

```
    return base_addr; // 返回基址
}
```

进程分配段时，首先检查是否已经分配，若未分配则执行分配过程。

分配时首先寻找第一个空闲内存块，对比程序段长和空闲块大小，小于空闲块空间时，分配内存，更新段表，更新空闲块大小，然后再对比剩余空间是否还能满足数据段需要，无法满足，则寻找下一个未被程序段占用的空闲块，若大小满足则分配，不满足继续查找下一个，更新段表，更新空闲块大小。

```
// 回收内存
void RecycleMem(int pid) {
    struct Program *index;
    index = pro_head;
    for (; index != NULL; index = index->next) {
        // 找到进程的节点
        if (index->pid == pid) {
            //找到程序段，数据段段号
            int main_seg_num = index->main_seg_num;
            int data_seg_num = index->data_seg_num;
            if (main_seg_num != -1 && data_seg_num != -1) {
                // 调用按段号回收内存
                RecycleSeg(main_seg_num); // 按照段号回收内存
                index->main_seg_num = -1; // 恢复成初始化
                RecycleSeg(data_seg_num); // 按照段号回收内存
                index->data_seg_num = -1; // 恢复成初始化
            }
        }
    }
}

// 段号回收内存
void RecycleSeg(int seg_num) {
    struct Segtable*item;
    item = seg_table_head;
    // 根据段号找到该段
    for (; item != NULL; item = item->next) {
        if (item->seg_num == seg_num) {
            break;
        }
    }
    //段表长度和基地址置初始化
    int base_addr;
    int seg_length;
    if (item != NULL) {
```

```
            base_addr = item->base_addr;
            seg_length = item->seg_length;
            item->base_addr = -1;
            item->seg_length = -1;
            // 空闲块
            struct Memory *idle;
            idle = (struct Memory*)malloc(sizeof(struct Memory));
            idle->addr = base_addr;
            idle->length = seg_length;
            // 插入
            struct Memory *index;
            index = mem_head;
            if (base_addr < index->addr) {
                //在头部插入
                idle->next = index;
                mem_head = idle;
            }else {
                for (; index != NULL; index = index->next) {
                    if (index->next == NULL) {
                        index->next = idle;
                        // 在尾部插入
                    }else {
                        int pre_addr = index->addr + index->length;
                        int next_addr = index->next->addr;
                        if (base_addr >= pre_addr && base_addr <= next_addr) {
                            // 找中间值，插入中间
                            idle->next = index->next;
                            index->next = idle;
                            break;
                        }
                    }
                }
            }

        }
        // 整理空闲区
        sort();
}

// 整理空闲区函数
void sort() {
        // 如果两个相邻的空闲区，首尾相接 ，合并
        struct Memory *index;
        index = mem_head;
```

```
    while (index != NULL && index->next != NULL) {
        // 前一块尾地址
        int tear_addr = index->addr + index->length;
        // 后一块首地址
        int head_addr = index->next->addr;
        if (tear_addr == head_addr) {
            struct Memory *tmp;
            tmp = index->next;
            // 长度合并
            index->length = index->length + tmp->length;
            // 指针转向
            index->next = tmp->next; // 跳过被合并的节点
            free(tmp);        // 删除节点
            continue;
        }
        // 连接下一个
        index = index->next;
    }
    index = NULL;// 清空
}
```

回收内存时，首先根据进程 id 遍历查找，找到之后，临时存储其基址和长度，将段结构初始化。根据基址决定出该空闲内存块需要插入的位置，将插入后的内存进行整理，相邻空闲区域合并。

7. 程序运行的主要界面和结果截图

界面：

申请内存：

创建进程：

```
请输入操作后回车:2
请输入要创建的进程ID,     程序段长度,      数据段长度
1
10
20
```

```
请输入操作后回车:2
请输入要创建的进程ID,     程序段长度,      数据段长度
2
30
40
```

```
请输入操作后回车:2
请输入要创建的进程ID,     程序段长度,      数据段长度
3
50
60
```

进程信息：

```
请输入操作后回车:5
**********************************进程信息**************************************
进程号          程序段长          程序段号          数据段长          数据段号
 3                50                -1                60                -1
 2                30                -1                40                -1
 1                10                -1                20                -1
********************************************************************************
**********************************段表信息**************************************
未分配段
********************************************************************************
```

进程调入：

```
请输入操作后回车:5
**********************************进程信息**************************************
进程号          程序段长          程序段号          数据段长          数据段号
 3                50                4                 60                5
 2                30                2                 40                3
 1                10                0                 20                1
********************************************************************************
**********************************段表信息**************************************
段号            基址              段长
0                0                 10
1                10                20
2                30                30
3                60                40
4                100               50
5                150               60
********************************************************************************
```

内存回收：

```
请输入操作后回车:4
请输入要释放的进程ID：2

**********分段存储模型**********
***  1、申请内存          2、创建进程        ***
***  3、进程调入          4、释放内存        ***
***  5、进程信息          6、可用内存信息    ***
***  7、保存数据到磁盘    8、加载磁盘数据    ***
***  9、退出                                 ***
*****************************************
请输入操作后回车:5
**********进程信息**********
进程号          程序段长          程序段号          数据段长          数据段号
 3                50                4                60                5
 2                30                -1                40                -1
 1                10                0                20                1
*****************************************************************************
**********段表信息**********
段号            基址              段长
0               0                 10
1               10                20
2               -1                -1
3               -1                -1
4               100               50
5               150               60
*****************************************************************************
```

证明内存分配正确性：

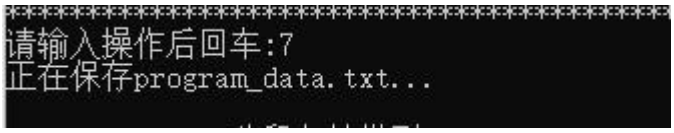
(申请 pid=4，程序段长 30，数据段长 60 的进程，调入)

```
请输入操作后回车:5
**********进程信息**********
进程号          程序段长          程序段号          数据段长          数据段号
 4                30                2                60                3
 3                50                4                60                5
 2                30                -1                40                -1
 1                10                0                20                1
*****************************************************************************
**********段表信息**********
段号            基址              段长
0               0                 10
1               10                20
2               30                30
3               210               60
4               100               50
5               150               60
*****************************************************************************
```

可用内存：

```
请输入操作后回车:6
*********可用内存信息**********
内存始址                可用长度
60                      40
270                     730
************************************
```

保存数据到磁盘：



8. 总结和感想体会

通过本次实验，我熟悉了段以及段表的结构，并且构建相应的数据结构，组成一个功能性的模型，其中利用到了大量链表操作，又去查阅之前学习过的数据结构内容，构建结构体并不难，难的是对其中的数据进行操作，必须要按照分段存储的要求来，要判断是否可以分配内存以及内存块是否够用，仅仅是分配过程就占了工程代码量的 1/4 左右，在释放时，回收内存空间甚至比分配还要复杂，要考虑到插入位置和内存块临近合并等。本次实验让我加深了对于段式存储的理解，明白了其工作流程，并且通过模型复现出基本功能，锻炼了实践能力，让我收获颇多。

参考文献

[1] 汤小丹、 梁红兵、 哲凤屏 、汤子瀛.计算机操作系统第四版[M].西安：西安电子科技大学出版社出版,2014.

[2]严蔚敏,吴伟民.数据结构-C 语言版[M].北京：清华大学大学出版社出版,2012.

附录